深入浅出Flink(3)

一、课前准备

掌握上节课内容

二、课堂主题

flink是一个有状态的流，本次课深入了解这个有状态的流

三、课程目标

1. 掌握State知识
2. 掌握Flink三种State Backend

四、知识要点

4.1 State

4.1.1 state概述

Apache Flink® — Stateful Computations over Data Streams

回顾单词计数的例子

```java
/**
 * 单词计数
 */
public class WordCount {
    public static void main(String[] args) throws Exception {
        StreamExecutionEnvironment env =
StreamExecutionEnvironment.getExecutionEnvironment();
        DataStreamSource<String> data = env.socketTextStream("localhost", 8888);
        SingleOutputStreamOperator<Tuple2<String, Integer>> result =
data.flatMap(new FlatMapFunction<String, Tuple2<String, Integer>>() {
            @Override
            public void flatMap(String line, Collector<Tuple2<String, Integer>>
collector) throws Exception {
                String[] fields = line.split(",");
                for (String word : fields) {
                    collector.collect(new Tuple2<>(word, 1));
                }
            }
        }).keyBy("0")
                .sum(1);

        result.print();

        env.execute("WordCount");
    }
}
```

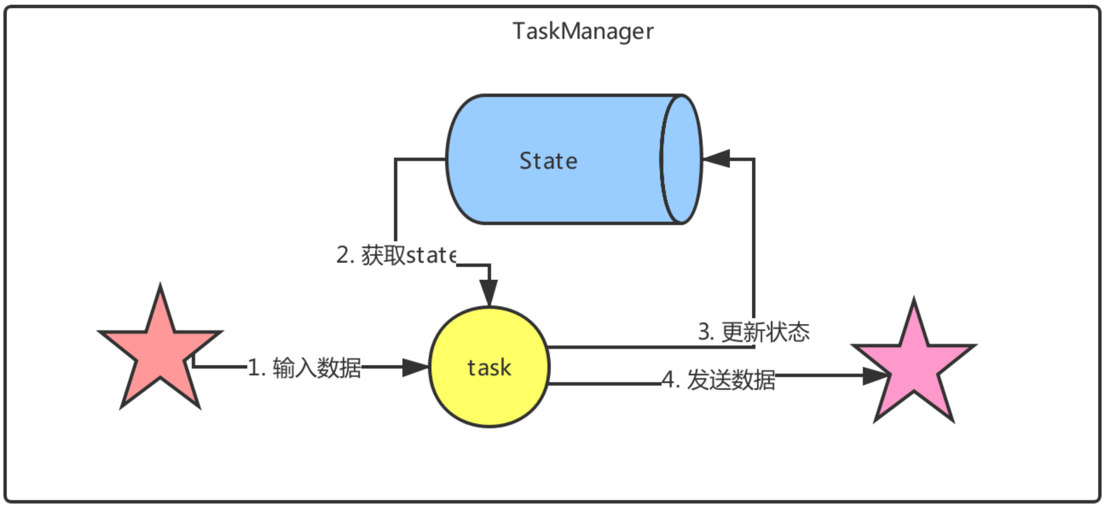
输入

```
hadoop,hadoop
hadoop
hive,hadoop
```

输出

```
4> (hadoop,1)
4> (hadoop,2)
4> (hadoop,3)
1> (hive,1)
4> (hadoop,4)
```

我们会发现，单词出现的次数有累计的效果。如果没有状态的管理，是不会有累计的效果的，所以Flink里面还有state的概念。

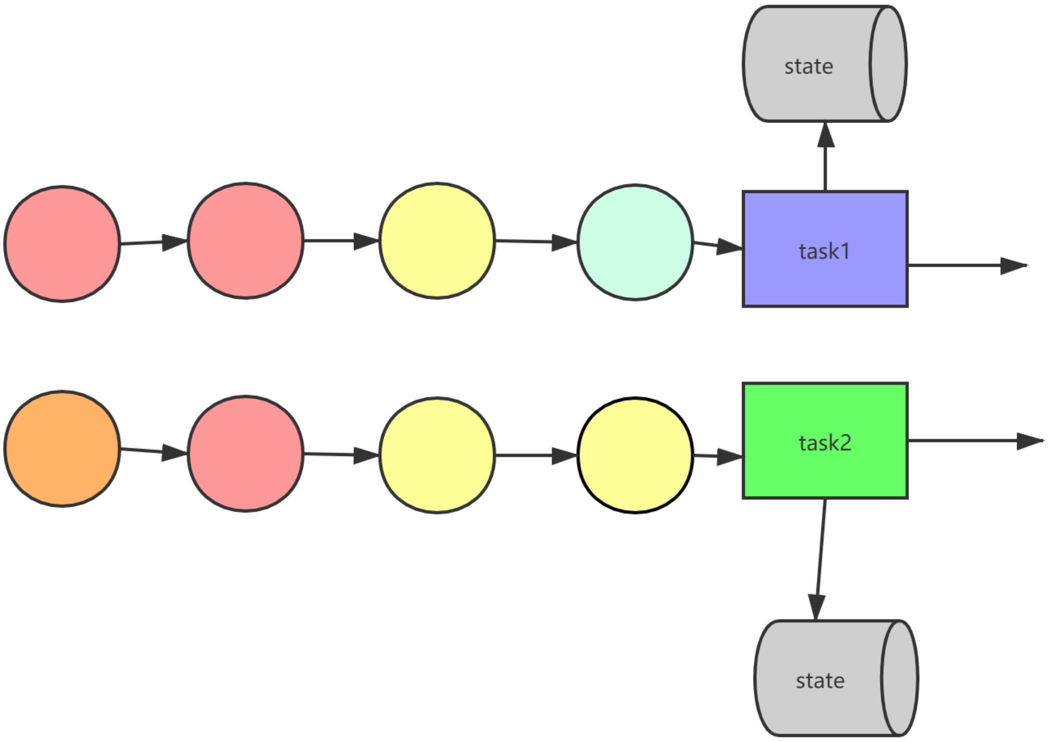


state：一般指一个具体的task/operator的状态。State可以被记录，在失败的情况下数据还可以恢复，Flink中有两种基本类型的State：Keyed State，Operator State，他们两种都可以以两种形式存在：原始状态(raw state)和托管状态(managed state).
托管状态：由Flink框架管理的状态，我们通常使用的就是这种。
原始状态：由用户自行管理状态具体的数据结构，框架在做checkpoint的时候，使用byte[]来读写状态内容，对其内部数据结构一无所知。通常在DataStream上的状态推荐使用托管的状态，当实现一个用户自定义的operator时，**会使用到原始状态**。但是我们工作中一般不常用，所以我们不考虑他。

4.1.2 State类型

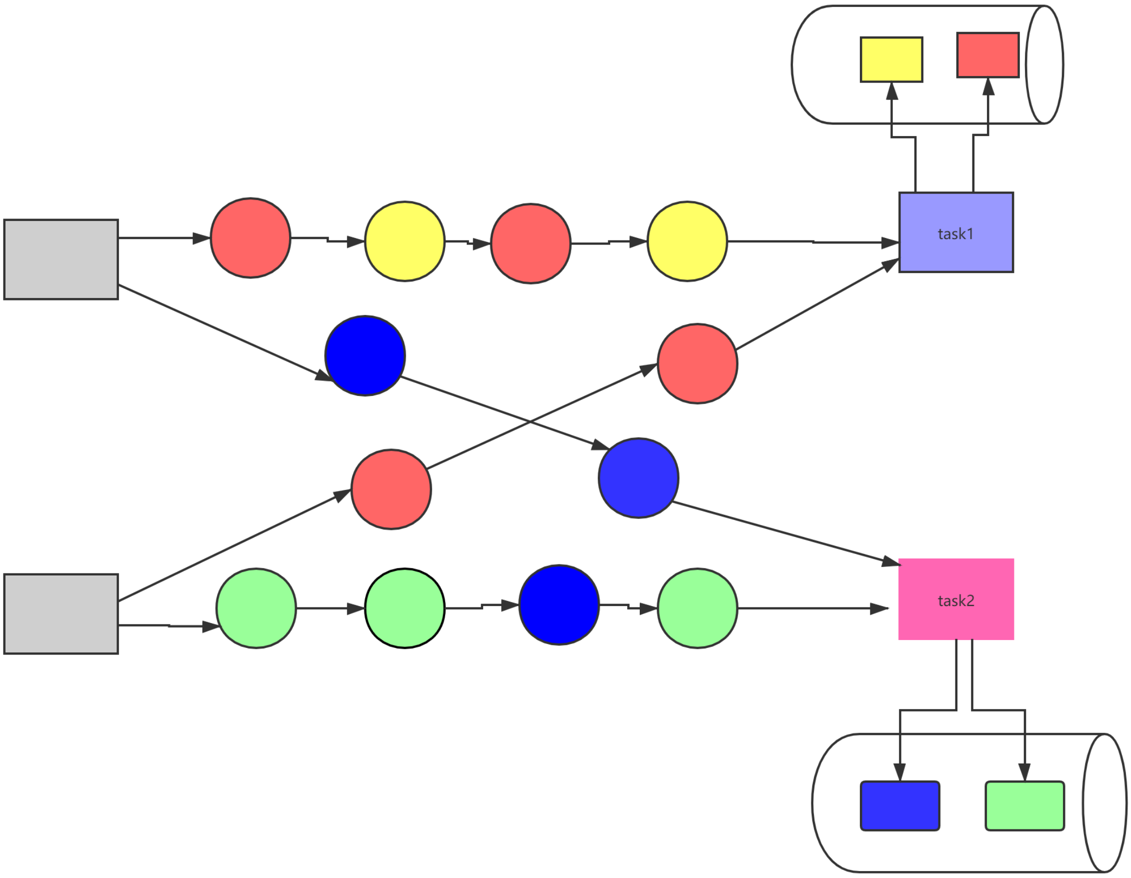
Operator State

1. operator



   state是task级别的state，说白了就是每个task对应一个state
2. Kafka Connector source中的每个分区（task）都需要记录消费的topic的partition和offset等信息。

Keyed State



1. keyed state 记录的是每个key的状态

2. Keyed state托管状态有五种类型：
    1. ValueState
    2. ListState
    3. MapState
    4. ReducingState
    5. AggregatingState

4.1.3 Keyed State的案例演示

ValueState

```
/**
 *  ValueState<T> ：这个状态为每一个 key 保存一个值
 *      value() 获取状态值
 *      update() 更新状态值
 *      clear() 清除状态
 */
public class CountWindowAverageWithValueState
        extends RichFlatMapFunction<Tuple2<Long, Long>, Tuple2<Long, Double>> {
    // 用以保存每个 key 出现的次数，以及这个 key 对应的 value 的总值
    // managed keyed state
    //1. ValueState 保存的是对应的一个 key 的一个状态值
    private ValueState<Tuple2<Long, Long>> countAndSum;

    @Override
    public void open(Configuration parameters) throws Exception {
        // 注册状态
        ValueStateDescriptor<Tuple2<Long, Long>> descriptor =
                new ValueStateDescriptor<Tuple2<Long, Long>>(
                        "average",  // 状态的名字
                        Types.TUPLE(Types.LONG, Types.LONG)); // 状态存储的数据类型
        countAndSum = getRuntimeContext().getState(descriptor);
    }

    @Override
    public void flatMap(Tuple2<Long, Long> element,
                        Collector<Tuple2<Long, Double>> out) throws Exception {
        // 拿到当前的 key 的状态值
        Tuple2<Long, Long> currentState = countAndSum.value();

        // 如果状态值还没有初始化，则初始化
        if (currentState == null) {
            currentState = Tuple2.of(0L, 0L);
        }

        // 更新状态值中的元素的个数
        currentState.f0 += 1;

        // 更新状态值中的总值
        currentState.f1 += element.f1;

        // 更新状态
        countAndSum.update(currentState);

        // 判断，如果当前的 key 出现了 3 次，则需要计算平均值，并且输出
        if (currentState.f0 >= 3) {
            double avg = (double)currentState.f1 / currentState.f0;
            // 输出 key 及其对应的平均值
            out.collect(Tuple2.of(element.f0, avg));
            //  清空状态值
            countAndSum.clear();
        }
    }
}




/**
 * 需求：当接收到的相同 key 的元素个数等于 3 个
 *  就计算这些元素的 value 的平均值。
 *  计算 keyed stream 中每 3 个元素的 value 的平均值

 每三个数求一次
 */
public class TestKeyedStateMain {
    public static void main(String[] args) throws  Exception{
        StreamExecutionEnvironment env =
StreamExecutionEnvironment.getExecutionEnvironment();

        DataStreamSource<Tuple2<Long, Long>> dataStreamSource =
                env.fromElements(Tuple2.of(1L, 3L), Tuple2.of(1L, 5L),
Tuple2.of(1L, 7L),
                        Tuple2.of(2L, 4L), Tuple2.of(2L, 2L), Tuple2.of(2L,
5L));

        // 输出：
        //(1,5.0)
        //(2,3.6666666666666665)
        dataStreamSource
                .keyBy(0)
                .flatMap(new CountWindowAverageWithValueState())
                .print();

        env.execute("TestStatefulApi");
    }
}
```

结果输出：

```
3> (1,5.0)
4> (2,3.6666666666666665)
```

ListState

```
/**
 *  ListState<T> ：这个状态为每一个 key 保存集合的值
 *      get() 获取状态值
 *      add() / addAll() 更新状态值，将数据放到状态中
 *      clear() 清除状态
 */
public class CountWindowAverageWithListState
```

```
        extends RichFlatMapFunction<Tuple2<Long, Long>, Tuple2<Long, Double>> {
    // managed keyed state
    //1. ListState 保存的是对应的一个 key 的出现的所有的元素
    private ListState<Tuple2<Long, Long>> elementsByKey;

    @Override
    public void open(Configuration parameters) throws Exception {
        // 注册状态
        ListStateDescriptor<Tuple2<Long, Long>> descriptor =
                new ListStateDescriptor<Tuple2<Long, Long>>(
                        "average",  // 状态的名字
                        Types.TUPLE(Types.LONG, Types.LONG)); // 状态存储的数据类型
        elementsByKey = getRuntimeContext().getListState(descriptor);
    }

    @Override
    public void flatMap(Tuple2<Long, Long> element,
                        Collector<Tuple2<Long, Double>> out) throws Exception {
        // 拿到当前的 key 的状态值
        Iterable<Tuple2<Long, Long>> currentState = elementsByKey.get();

        // 如果状态值还没有初始化，则初始化
        if (currentState == null) {
            elementsByKey.addAll(Collections.emptyList());
        }

        // 更新状态
        elementsByKey.add(element);

        // 判断，如果当前的 key 出现了 3 次，则需要计算平均值，并且输出
        List<Tuple2<Long, Long>> allElements =
Lists.newArrayList(elementsByKey.get());
        if (allElements.size() >= 3) {
            long count = 0;
            long sum = 0;
            for (Tuple2<Long, Long> ele : allElements) {
                count++;
                sum += ele.f1;
            }
            double avg = (double) sum / count;
            out.collect(Tuple2.of(element.f0, avg));

            // 清除状态
            elementsByKey.clear();
        }
    }
}


/**
 * 需求：当接收到的相同 key 的元素个数等于 3 个或者超过 3 个的时候
 *  就计算这些元素的 value 的平均值。
 *  计算 keyed stream 中每 3 个元素的 value 的平均值
 */
public class TestKeyedStateMain {
    public static void main(String[] args) throws  Exception{
        StreamExecutionEnvironment env =
StreamExecutionEnvironment.getExecutionEnvironment();
```

```
        DataStreamSource<Tuple2<Long, Long>> dataStreamSource =
                env.fromElements(Tuple2.of(1L, 3L), Tuple2.of(1L, 5L),
Tuple2.of(1L, 7L),
                        Tuple2.of(2L, 4L), Tuple2.of(2L, 2L), Tuple2.of(2L,
5L));

        // 输出:
        //(1,5.0)
        //(2,3.6666666666666665)
        dataStreamSource
                .keyBy(0)
                .flatMap(new CountWindowAverageWithListState())
                .print();

        env.execute("TestStatefulApi");
    }
}
```

结果输出:

```
3> (1,5.0)
4> (2,3.6666666666666665)
```

MapState

```
/**
 *  MapState<K, V> : 这个状态为每一个 key 保存一个 Map 集合
 *      put() 将对应的 key 的键值对放到状态中
 *      values() 拿到 MapState 中所有的 value
 *      clear() 清除状态
 */
public class CountWindowAverageWithMapState
        extends RichFlatMapFunction<Tuple2<Long, Long>, Tuple2<Long, Double>> {
    // managed keyed state
    //1. MapState : key 是一个唯一的值，value 是接收到的相同的 key 对应的 value 的值
    private MapState<String, Long> mapState;

    @Override
    public void open(Configuration parameters) throws Exception {
        // 注册状态
        MapStateDescriptor<String, Long> descriptor =
                new MapStateDescriptor<String, Long>(
                        "average",  // 状态的名字
                        String.class, Long.class); // 状态存储的数据类型
        mapState = getRuntimeContext().getMapState(descriptor);
    }

    @Override
    public void flatMap(Tuple2<Long, Long> element,
                        Collector<Tuple2<Long, Double>> out) throws Exception {
        mapState.put(UUID.randomUUID().toString(), element.f1);

        // 判断，如果当前的 key 出现了 3 次，则需要计算平均值，并且输出
        List<Long> allElements = Lists.newArrayList(mapState.values());
        if (allElements.size() >= 3) {
            long count = 0;
            long sum = 0;
            for (Long ele : allElements) {
                count++;
                sum += ele;
            }
            double avg = (double) sum / count;
            out.collect(Tuple2.of(element.f0, avg));

            // 清除状态
            mapState.clear();
        }
    }
}


/**
 * 需求：当接收到的相同 key 的元素个数等于 3 个或者超过 3 个的时候
 *  就计算这些元素的 value 的平均值。
 *  计算 keyed stream 中每 3 个元素的 value 的平均值
 */
public class TestKeyedStateMain {
    public static void main(String[] args) throws  Exception{
        StreamExecutionEnvironment env =
StreamExecutionEnvironment.getExecutionEnvironment();

        DataStreamSource<Tuple2<Long, Long>> dataStreamSource =
                env.fromElements(Tuple2.of(1L, 3L), Tuple2.of(1L, 5L),
Tuple2.of(1L, 7L),
                        Tuple2.of(2L, 4L), Tuple2.of(2L, 2L), Tuple2.of(2L,
5L));

        // 输出：
        //(1,5.0)
        //(2,3.6666666666666665)
        dataStreamSource
                .keyBy(0)
                .flatMap(new CountWindowAverageWithMapState())
                .print();

        env.execute("TestStatefulApi");
    }
}
```

输出结果：

4> (2,3.6666666666666665)
3> (1,5.0)

ReducingState

```
/**
 *  ReducingState<T> ：这个状态为每一个 key 保存一个聚合之后的值
 *      get() 获取状态值
 *      add()  更新状态值，将数据放到状态中
 *       clear() 清除状态
 */
public class SumFunction
        extends RichFlatMapFunction<Tuple2<Long, Long>, Tuple2<Long, Long>> {
    // managed keyed state
    // 用于保存每一个 key 对应的 value 的总值
    private ReducingState<Long> sumState;

    @Override
    public void open(Configuration parameters) throws Exception {
        // 注册状态
        ReducingStateDescriptor<Long> descriptor =
                new ReducingStateDescriptor<Long>(
                        "sum",  // 状态的名字
                        new ReduceFunction<Long>() { // 聚合函数
                            @Override
                            public Long reduce(Long value1, Long value2) throws
Exception {
                                return value1 + value2;
                            }
                        }, Long.class); // 状态存储的数据类型
        sumState = getRuntimeContext().getReducingState(descriptor);
    }

    @Override
    public void flatMap(Tuple2<Long, Long> element,
                        Collector<Tuple2<Long, Long>> out) throws Exception {
        // 将数据放到状态中
        sumState.add(element.f1);

        out.collect(Tuple2.of(element.f0, sumState.get()));
    }
}


public class TestKeyedStateMain2 {
    public static void main(String[] args) throws  Exception{
        StreamExecutionEnvironment env =
StreamExecutionEnvironment.getExecutionEnvironment();

        DataStreamSource<Tuple2<Long, Long>> dataStreamSource =
                env.fromElements(Tuple2.of(1L, 3L), Tuple2.of(1L, 5L),
Tuple2.of(1L, 7L),
                        Tuple2.of(2L, 4L), Tuple2.of(2L, 2L), Tuple2.of(2L,
5L));

        // 输出:
        //(1,5.0)
        //(2,3.6666666666666665)
        dataStreamSource
                .keyBy(0)
                .flatMap(new SumFunction())
                .print();

        env.execute("TestStatefulApi");
    }
}
```

输出:

4> (2,4)
4> (2,6)
4> (2,11)
3> (1,3)
3> (1,8)
3> (1,15)

AggregatingState

```
public class ContainsValueFunction
        extends RichFlatMapFunction<Tuple2<Long, Long>, Tuple2<Long, String>> {

    private AggregatingState<Long, String> totalStr;

    @Override
    public void open(Configuration parameters) throws Exception {
        // 注册状态
        AggregatingStateDescriptor<Long, String, String> descriptor =
                new AggregatingStateDescriptor<Long, String, String>(
                        "totalStr",  // 状态的名字
                        new AggregateFunction<Long, String, String>() {
                            @Override
                            public String createAccumulator() {
                                return "Contains: ";
                            }

                            @Override
                            public String add(Long value, String accumulator) {
                                if ("Contains: ".equals(accumulator)) {
                                    return accumulator + value;
                                }
                                return accumulator + " and " + value;
                            }

                            @Override
                            public String getResult(String accumulator) {
                                return accumulator;
                            }

                            @Override
                            public String merge(String a, String b) {
                                return a + " and " + b;
                            }
                        }, String.class); // 状态存储的数据类型
        totalStr = getRuntimeContext().getAggregatingState(descriptor);
    }

    @Override
    public void flatMap(Tuple2<Long, Long> element,
                        Collector<Tuple2<Long, String>> out) throws Exception {
        totalStr.add(element.f1);
        out.collect(Tuple2.of(element.f0, totalStr.get()));
    }
}
```

```java
public class TestKeyedStateMain2 {
    public static void main(String[] args) throws  Exception{
        StreamExecutionEnvironment env =
StreamExecutionEnvironment.getExecutionEnvironment();

        DataStreamSource<Tuple2<Long, Long>> dataStreamSource =
                env.fromElements(Tuple2.of(1L, 3L), Tuple2.of(1L, 5L),
Tuple2.of(1L, 7L),
                        Tuple2.of(2L, 4L), Tuple2.of(2L, 2L), Tuple2.of(2L,
5L));


        dataStreamSource
                .keyBy(0)
                .flatMap(new ContainsValueFunction())
                .print();

        env.execute("TestStatefulApi");
    }
}
```

输出：

```
4> (2,Contains：4)
3> (1,Contains：3)
3> (1,Contains：3 and 5)
3> (1,Contains：3 and 5 and 7)
4> (2,Contains：4 and 2)
4> (2,Contains：4 and 2 and 5)
```

4.1.4 KeyedState 案例演示

需求：将两个流中，订单号一样的数据合并在一起输出

orderinfo1数据 topic

商品平台

```
123,拖把,30.0
234,牙膏,20.0
345,被子,114.4
333,杯子,112.2
444,Mac电脑,30000.0
```

orderinfo2数据 topic

```
123,2020-11-11 10:11:12,江苏
234,2020-11-11 11:11:13,云南
345,2020-11-11 12:11:14,安徽
333,2020-11-11 13:11:15,北京
444,2020-11-11 14:11:16,深圳
```

代码实现：

```
public class Constants {
    public static final String
ORDER_INFO1_PATH="D:\\nx\\flinklesson\\src\\main\\input\\OrderInfo1.txt";
    public static final String
ORDER_INFO2_PATH="D:\\nx\\flinklesson\\src\\main\\input\\OrderInfo2.txt";
}
```

```
public class OrderInfo1 {
    //订单ID
    private Long orderId;
    //商品名称
    private String productName;
    //价格
    private Double price;

   public OrderInfo1(){

   }

   public OrderInfo1(Long orderId,String productName,Double price){
       this.orderId=orderId;
       this.productName=productName;
       this.price=price;
   }

    @Override
    public String toString() {
        return "OrderInfo1{" +
                "orderId=" + orderId +
                ", productName='" + productName + '\'' +
                ", price=" + price +
                '}';
    }

    public Long getOrderId() {
        return orderId;
    }

    public void setOrderId(Long orderId) {
        this.orderId = orderId;
    }
    public String getProductName() {
        return productName;
    }

    public void setProductName(String productName) {
        this.productName = productName;
    }

    public Double getPrice() {
        return price;
    }

    public void setPrice(Double price) {
        this.price = price;
    }

    public static OrderInfo1 string2OrderInfo1(String line){
        OrderInfo1 orderInfo1 = new OrderInfo1();
        if(line != null && line.length() > 0){
           String[] fields = line.split(",");
            orderInfo1.setOrderId(Long.parseLong(fields[0]));
            orderInfo1.setProductName(fields[1]);
            orderInfo1.setPrice(Double.parseDouble(fields[2]));
       }
       return orderInfo1;
    }
}
```

```
public class OrderInfo2 {
    //订单ID
    private Long orderId;
    //下单时间
    private String orderDate;
    //下单地址
    private String address;

    public OrderInfo2(){

    }
    public OrderInfo2(Long orderId,String orderDate,String address){
        this.orderId = orderId;
        this.orderDate = orderDate;
        this.address = address;
    }

    @Override
    public String toString() {
        return "OrderInfo2{" +
                "orderId=" + orderId +
                ", orderDate='" + orderDate + '\'' +
                ", address='" + address + '\'' +
                '}';
    }

    public Long getOrderId() {
        return orderId;
```

```java
    }

    public void setOrderId(Long orderId) {
        this.orderId = orderId;
    }

    public String getOrderDate() {
        return orderDate;
    }

    public void setOrderDate(String orderDate) {
        this.orderDate = orderDate;
    }

    public String getAddress() {
        return address;
    }

    public void setAddress(String address) {
        this.address = address;
    }


    public static OrderInfo2 string2OrderInfo2(String line){
        OrderInfo2 orderInfo2 = new OrderInfo2();
        if(line != null && line.length() > 0){
            String[] fields = line.split(",");
            orderInfo2.setOrderId(Long.parseLong(fields[0]));
            orderInfo2.setOrderDate(fields[1]);
            orderInfo2.setAddress(fields[2]);
        }

        return orderInfo2;
    }
}
```

```java
/**
 * 自定义source
 */
public class FileSource implements SourceFunction<String> {
    //文件路径
    public String filePath;
    public FileSource(String filePath){
        this.filePath = filePath;
    }

    private InputStream inputStream;
    private BufferedReader reader;

    private Random random = new Random();

    @Override
    public void run(SourceContext<String> ctx) throws Exception {
           reader = new BufferedReader(new InputStreamReader(new
FileInputStream(filePath)));
           String line = null;
           while ((line = reader.readLine()) != null) {
               // 模拟发送数据
               TimeUnit.MILLISECONDS.sleep(random.nextInt(500));
               // 发送数据
               ctx.collect(line);
           }
        if(reader != null){
            reader.close();
        }
        if(inputStream != null){
            inputStream.close();
        }

    }

    @Override
    public void cancel()  {
      try{
          if(reader != null){
              reader.close();
          }
          if(inputStream != null){
              inputStream.close();
          }
      }catch (Exception e){

      }
    }
}
```

```
public class OrderStream {
    public static void main(String[] args) throws  Exception {
        StreamExecutionEnvironment env =
StreamExecutionEnvironment.getExecutionEnvironment();
        DataStreamSource<String> info1 = env.addSource(new
FileSource(Constants.ORDER_INFO1_PATH));
        DataStreamSource<String> info2 = env.addSource(new
FileSource(Constants.ORDER_INFO2_PATH));

        KeyedStream<OrderInfo1, Long> orderInfo1Stream = info1.map(line ->
string2OrderInfo1(line))
                .keyBy(orderInfo1 -> orderInfo1.getOrderId());

        KeyedStream<OrderInfo2, Long> orderInfo2Stream = info2.map(line ->
string2OrderInfo2(line))
                .keyBy(orderInfo2 -> orderInfo2.getOrderId());

        orderInfo1Stream.connect(orderInfo2Stream)
                .flatMap(new EnrichmentFunction())
                .print();
        env.execute("OrderStream");

    }

    /**
     *   IN1，第一个流的输入的数据类型
         IN2，第二个流的输入的数据类型
         OUT，输出的数据类型
     */
    public static class EnrichmentFunction extends

 RichCoFlatMapFunction<OrderInfo1,OrderInfo2,Tuple2<OrderInfo1,OrderInfo2>>{
        //定义第一个流 key对应的state
        private ValueState<OrderInfo1> orderInfo1State;
        //定义第二个流 key对应的state
        private ValueState<OrderInfo2> orderInfo2State;

        @Override
        public void open(Configuration parameters) {
            orderInfo1State = getRuntimeContext()
                    .getState(new ValueStateDescriptor<OrderInfo1>("info1",
OrderInfo1.class));
            orderInfo2State = getRuntimeContext()
                    .getState(new ValueStateDescriptor<OrderInfo2>
("info2",OrderInfo2.class));

        }

        @Override
        public void flatMap1(OrderInfo1 orderInfo1, Collector<Tuple2<OrderInfo1,
OrderInfo2>> out) throws Exception {
            OrderInfo2 value2 = orderInfo2State.value();
            if(value2 != null){
                orderInfo2State.clear();
                out.collect(Tuple2.of(orderInfo1,value2));
            }else{
                orderInfo1State.update(orderInfo1);
            }

        }

        @Override
        public void flatMap2(OrderInfo2 orderInfo2, Collector<Tuple2<OrderInfo1,
OrderInfo2>> out)throws Exception {
            OrderInfo1 value1 = orderInfo1State.value();
            if(value1 != null){
                orderInfo1State.clear();
                out.collect(Tuple2.of(value1,orderInfo2));
            }else{
                orderInfo2State.update(orderInfo2);
            }

        }
    }
}
```

4.1.5 Operator State案例演示

ListState

```
public class CustomSink
        implements SinkFunction<Tuple2<String, Integer>>, CheckpointedFunction {

    // 用于缓存结果数据的
    private List<Tuple2<String, Integer>> bufferElements;
    // 表示内存中数据的大小阈值
    private int threshold;
    // 用于保存内存中的状态信息
    private ListState<Tuple2<String, Integer>> checkpointState;
    // StateBackend
    // checkpoint

    public CustomSink(int threshold) {
        this.threshold = threshold;
        this.bufferElements = new ArrayList<>();
    }

    @Override
    public void invoke(Tuple2<String, Integer> value, Context context) throws
Exception {
        // 可以将接收到的每一条数据保存到任何的存储系统中
        bufferElements.add(value);
        if (bufferElements.size() == threshold) {
            // 简单打印
            System.out.println("自定义格式: " + bufferElements);
            bufferElements.clear();
        }
    }

    // 用于将内存中数据保存到状态中
    @Override
    public void snapshotState(FunctionSnapshotContext context) throws Exception
{
        checkpointState.clear();
        for (Tuple2<String, Integer> ele : bufferElements) {
            checkpointState.add(ele);
        }
    }
    // 用于在程序挥发的时候从状态中恢复数据到内存
    @Override
    public void initializeState(FunctionInitializationContext context) throws
Exception {
        ListStateDescriptor<Tuple2<String, Integer>> descriptor =
                new ListStateDescriptor<Tuple2<String, Integer>>(
                        "bufferd -elements",
                        TypeInformation.of(new TypeHint<Tuple2<String, Integer>>
() {}));
        // 注册一个 operator state
        checkpointState =
context.getOperatorStateStore().getListState(descriptor);

        if (context.isRestored()) {
            for (Tuple2<String, Integer> ele : checkpointState.get()) {
                bufferElements.add(ele);
            }
        }
    }
}


/**
 * 需求: 每两条数据打印一次结果 1000
 */
public class TestOperatorStateMain {
    public static void main(String[] args) throws  Exception{
        StreamExecutionEnvironment env =
StreamExecutionEnvironment.getExecutionEnvironment();

        DataStreamSource<Tuple2<String, Integer>> dataStreamSource =
                env.fromElements(Tuple2.of("Spark", 3), Tuple2.of("Hadoop", 5),
Tuple2.of("Hadoop", 7),
                        Tuple2.of("Spark", 4));


        dataStreamSource
                .addSink(new CustomSink(2)).setParallelism(1);

        env.execute("TestStatefulApi");
    }
}
```

输出结果：

自定义格式：[(Spark,3), (Hadoop,5)]

自定义格式：[(Hadoop,7), (Spark,4)]

BroadCastStete案例演示

```
/**
 * 数据流:
 * my love flink
 * 控制流:
 * key  flink  -> 代表数据流里面，只要包含flink的单词才会被打印出来。
 */
public class TestBroadcastState {
    public static void main(String[] args) throws Exception {
        //获取执行环境
        StreamExecutionEnvironment env =
StreamExecutionEnvironment.getExecutionEnvironment();

        // 数据流
        DataStreamSource<String> dataStreamSource =
                env.socketTextStream("192.168.123.102", 9999);

        // 控制流
        DataStreamSource<String> controlStreamSource =
                env.socketTextStream("192.168.123.102", 8888);
        // 解析控制流中的数据成二元组
        DataStream<Tuple2<String, String>> controlStream =
controlStreamSource.map(new MapFunction<String, Tuple2<String, String>>() {
            @Override
            public Tuple2<String, String> map(String s) throws Exception {
                String[] strings = s.split(" ");
                return Tuple2.of(strings[0], (strings[1]));
            }
        });

        //broadcast控制流里面的数据
        MapStateDescriptor<String, String> descriptor = new
MapStateDescriptor<String, String>(
                "ControlStream",
                String.class,
                String.class
        );

        BroadcastStream<Tuple2<String, String>> broadcastStream =
                controlStream.broadcast(descriptor);

        // 将数据流和控制流进行连接，利用控制流中的数据来控制字符串的长度
        dataStreamSource
                .connect(broadcastStream)
                .process(new KeyWordsCheckProcessor())
                .print();


        env.execute("TestBroadcastState");
    }

    private static class KeyWordsCheckProcessor
            extends BroadcastProcessFunction<String, Tuple2<String, String>,
String> {

        MapStateDescriptor<String, String> descriptor =
                new MapStateDescriptor<String, String>(
                        "ControlStream",
                        String.class,
                        String.class
                );

        @Override
        public void processBroadcastElement(Tuple2<String, String> value,
                                            Context ctx, Collector<String> out)
throws Exception {
            // 将接收到的控制数据放到 broadcast state 中
            ctx.getBroadcastState(descriptor).put(value.f0, value.f1);
            //打印控制信息
            System.out.println(Thread.currentThread().getName() + " 接收到控制信息
:  " + value);
        }

        @Override
        public void processElement(String value,
                                   ReadOnlyContext ctx, Collector<String> out)
throws Exception {
            // 从 broadcast state 中拿到控制信息
            String keywords = ctx.getBroadcastState(descriptor).get("key");
            //获取符合条件的单词
            if (value.contains(keywords)) {
                out.collect(value);
            }
        }
    }
}
```

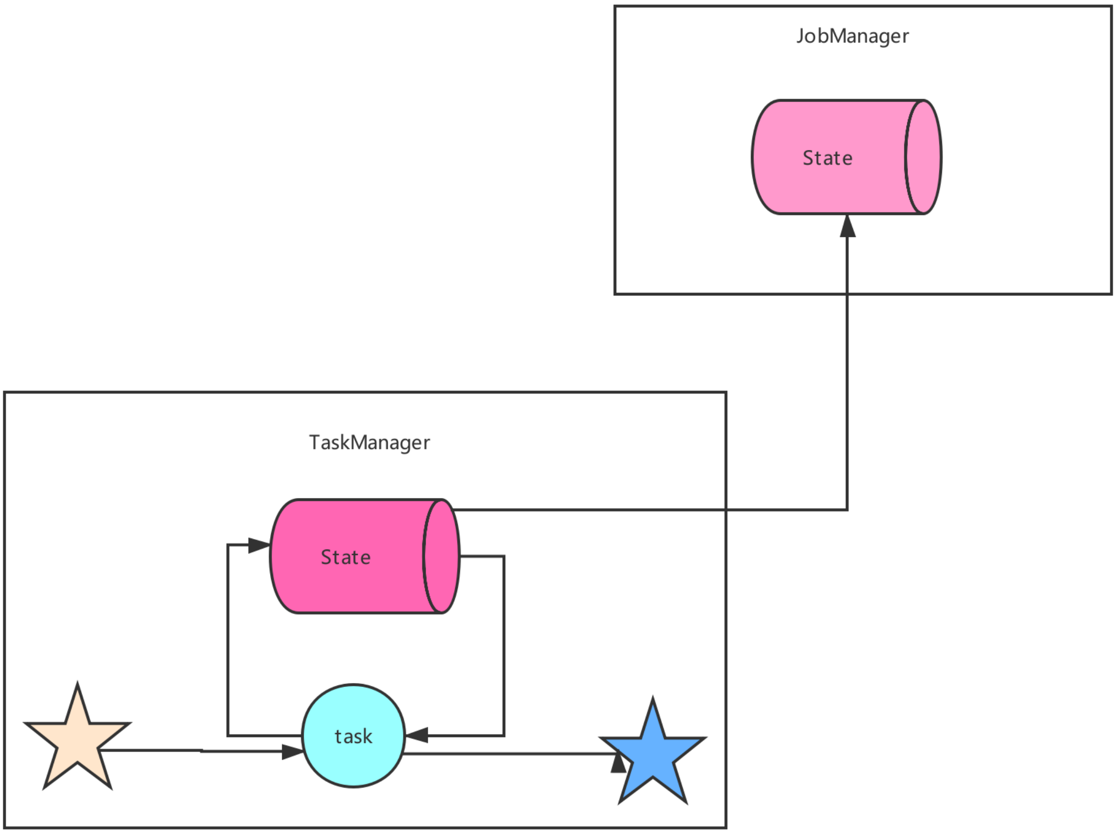
4.2 State backend

4.2.1 概述

Flink支持的StateBackend:

- *MemoryStateBackend* 默认的state的类型就是这种
- *FsStateBackend*
- *RocksDBStateBackend*

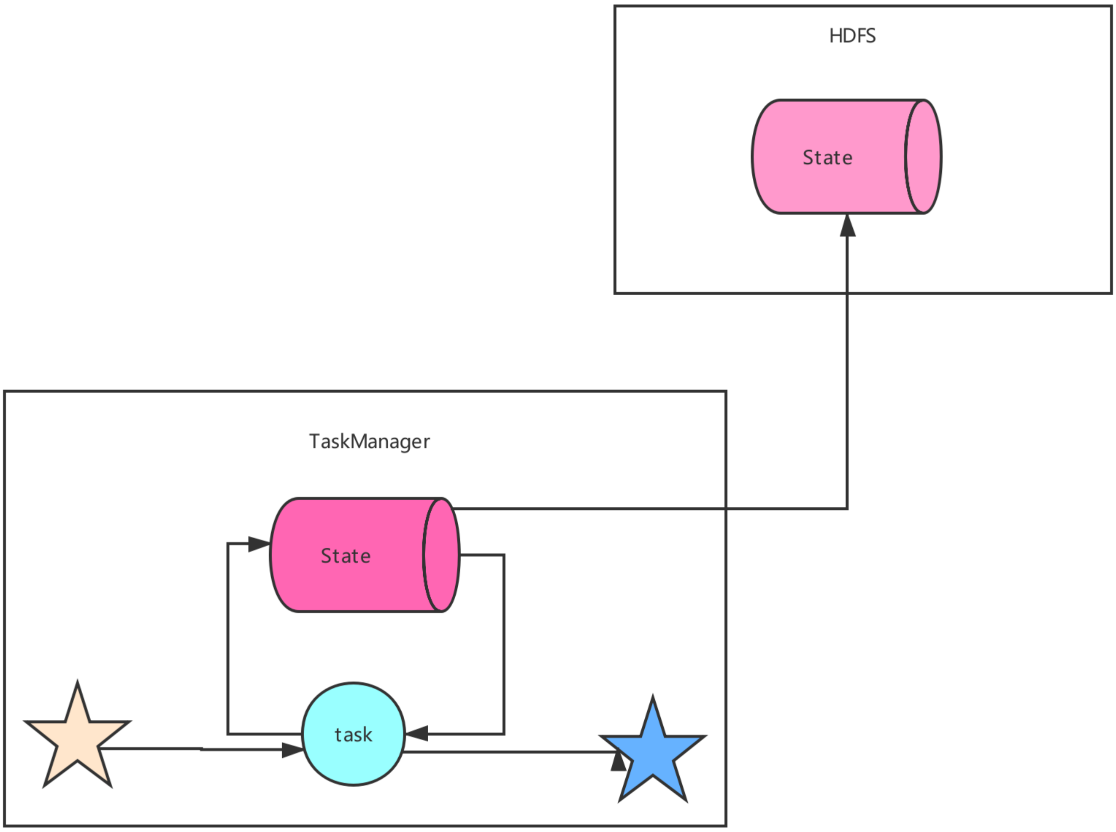
4.2.2 MemoryStateBackend



默认情况下，状态信息是存储在 TaskManager 的堆内存中的，checkpoint 的时候将状态保存到 JobManager 的堆内存中。

```
缺点：
•    只能保存数据量小的状态
•    状态数据有可能会丢失
优点：
•    开发测试很方便
```

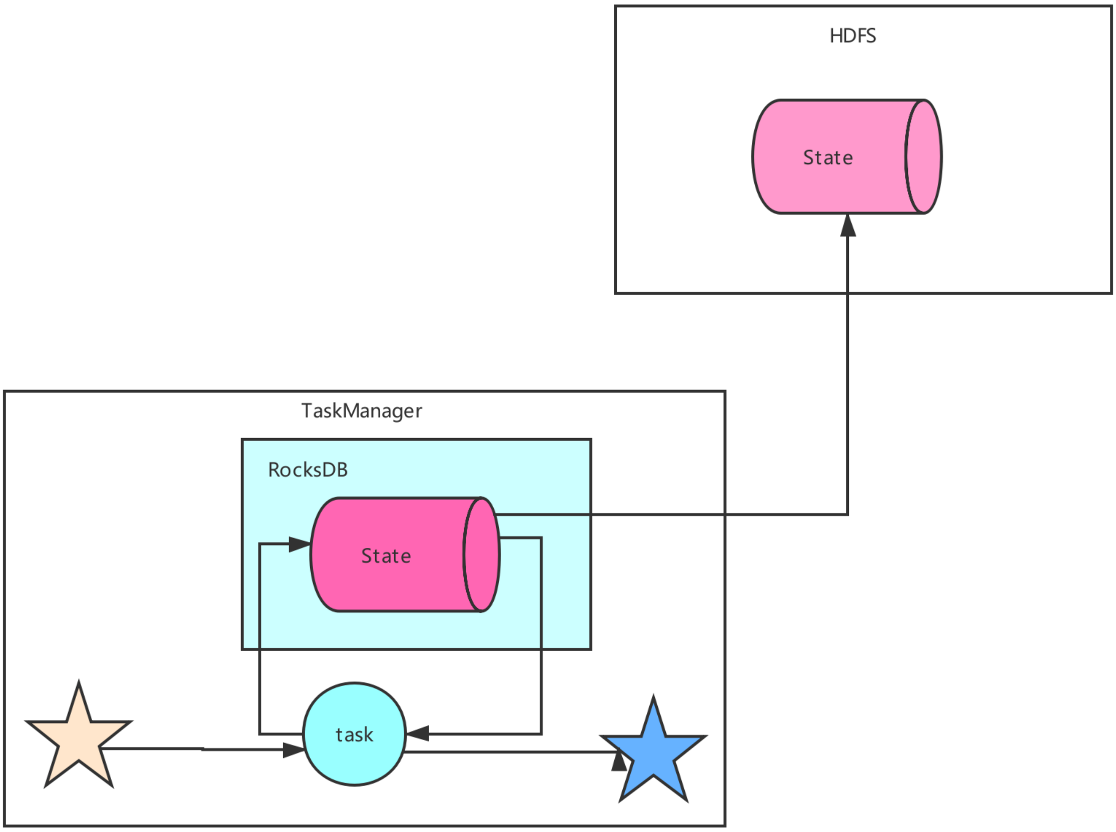
4.2.3 FSStateBackend



状态信息存储在 TaskManager 的堆内存中的，checkpoint 的时候将状态保存到指定的文件中 (HDFS 等文件系统)

```
缺点：
	状态大小受TaskManager内存限制(默认支持5M)
优点：
	状态访问速度很快
	状态信息不会丢失
用于： 生产，也可存储状态数据量大的情况
```

4.2.4 RocksDBStateBackend



状态信息存储在 RocksDB 数据库 (key-value 的数据存储服务)， 最终保存在本地文件中
checkpoint 的时候将状态保存到指定的文件中 (HDFS 等文件系统)

```
缺点：
    状态访问速度有所下降
优点：
    可以存储超大量的状态信息
    状态信息不会丢失
用于： 生产，可以存储超大量的状态信息
```

4.2.5 StateBackend配置方式

（1）单任务调整

```
修改当前任务代码
env.setStateBackend(new
FsStateBackend("hdfs://namenode:9000/flink/checkpoints"));
或者new MemoryStateBackend()
或者new RocksDBStateBackend(filebackend, true);【需要添加第三方依赖】
```

（2）全局调整(不建议)

```
修改flink-conf.yaml
state.backend: filesystem
state.checkpoints.dir: hdfs://namenode:9000/flink/checkpoints
注意：state.backend的值可以是下面几种：jobmanager(MemoryStateBackend),
filesystem(FsStateBackend), rocksdb(RocksDBStateBackend)
```

五、总结（5分钟）

1. State的类型

六 、作业

1. 掌握课上案例，为后面实践做准备

七 、互动